HUMAN-COMPUTER INTERACTION, 1986, Volume 2, pp. 135-166

Learning Flow of Control: Recursive and Iterative Procedures

Claudius M. Kessler and **John R. Anderson**
Carnegie-Mellon University

ABSTRACT

Two experiments were performed to study students' ability to write recursive and iterative programs and transfer between these two skills. Subjects wrote functions to accumulate instances into a list. Problems varied in terms of whether they were recursive or iterative, whether they operated on lists or numbers, whether they accumulated results in forward or backward manner, whether they accumulated on success or failure, and whether they simply skipped or ejected on failure to accumulate. Subjects had real difficulty only with the dimensions concerned with flow of control, namely, recursive versus iterative, and skip versus eject. We found positive transfer from writing iterative functions to writing recursive functions, but not vice versa. A subsequent protocol study revealed subjects had such a poor mental model of recursion that they developed poor learning strategies which hindered their understanding of iteration. It is argued that having an adequate model of the functionality of programming is prerequisite to learning to program, and that it is sensible pedagogical practice to base understanding of recursive flow of control on understanding iterative flow of control.

Authors' present addresses: Claudius M. Kessler and John R. Anderson, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213.

CONTENTS

1. INTRODUCTION: PREVIOUS EXPERIMENTS ON RECURSION AND ITERATION

Over the last few years there have been many attempts to foster and facilitate the acquisition of programming skills. A central research topic within this context is the issue of how novices acquire knowledge about control structures. For programming languages like Pascal, understanding the different kinds of iteration and their appropriate uses presents a major problem to the beginning student. A major problem when teaching novices a language like LISP is to find an effective way to teach recursion. Most students seem to have little trouble acquiring the language until recursive programming constructs are introduced. At this point, many students exhibit unusually strong comprehension problems that jeopardize their further progress.

Previous work comparing the acquisition of recursive and iterative procedures has shown that novices tend to understand the recursive process as an iterative one. Kahney and Eisenstadt (1982), for example, let novices and experts judge recursive programs. The subjects had to decide if the programs actually did what they were supposed to do and give an explanation of their answers. Nearly all of the novices' answers gave evidence of a wrong or inadequate model of recursion. The most popular misconception was to represent recursion as a loop structure that iterates through a problem.

Kurland and Pea (1983) showed that the same kind of misconception underlies children's understanding of recursive LOGO programs. Although subjects could mentally trace a tail-recursive procedure, they could not evaluate pro-

grams that included embedded recursive calls. The children's verbal protocols of their evaluation processes clearly showed that they viewed recursion as iteration.

Anzai and Uesato (1982) got subjects to write simple recursive constructs with about 1 hr of instruction. Subjects were taught a paper-and-pencil method to simulate a factorial function and then had to build factorial functions of their own. One group started with a recursive procedure and then switched to an iterative one, while a second group received the opposite order. The results show that writing iterative functions first facilitated the writing of recursive functions more than writing recursive functions first facilitated the writing of iterative functions. It was difficult for the subjects in the experiment to interpret recursive calls in the recursive procedures. Anzai and Uesato hypothesized that learning iteration first helps with recursion because iteration provides a model for what a recursive call does. However, learning recursion first does not help with iteration because iteration is easily mastered, and the recursive construct is not understood well enough to serve as the basis for transfer.

Unfortunately, Anzai and Uesato did not run control conditions that would allow them to distinguish the amount of facilitation due to increased practice from the effect of transferring from one procedure to another. It may be that subjects improve in the iteration-to-recursion condition because of general practice. In the recursion-to-iteration condition, there could be negative transfer combined with the effects of general practice. These possible explanations could be investigated if the transfer results of their conditions are compared with the improvement that would occur in conditions that transfer from recursion to further recursion and from iteration to further iteration.

With respect to the relevance of this study to computer programming, it should be noted that the subjects in the Anzai and Uesato experiment did not have to learn a computer language to execute the recursive procedures. The fact that one normally learns about recursion in the context of learning a computer language might introduce further factors in the process of skill acquisition that are not addressed by Anzai and Uesato.

The experiments reported in this article are designed to further investigate how novices learn about recursion and iteration and to test their ability to transfer from learning one construct to learning the other. Because we are interested in the acquisition of programming skills, our subjects learned about these constructs within the context of a programming language.

2. EXPERIMENT 1: NOVICE DIFFICULTIES WITH RECURSION AND ITERATION

The programming language used in this experiment, SIMPLE (Shrager & Pirolli, 1983), was designed especially for the purpose of studying the acquisition of recursive programming skills. SIMPLE does a few LISP-like operations

whose implementations read as close to English as possible. Subjects learned this language and then solved a set of problems in a recursive and/or iterative fashion.

The tasks were designed to address the following questions:

- Is it easier to learn iteration than it is to learn recursion?
- Is there transfer from learning one of these constructs to learning the other?
- What factors influence the difficulty of writing recursive or iterative functions?

2.1. Method

The Programming Problems. The programming tasks specified a set of constraints to extract certain book titles from a catalog. Each title was associated with a catalog number and a subject area.

The functions that had to be written differed on five dimensions (in parentheses, we give the name of the dimension as it will be used throughout this article):

1. whether they were recursive or iterative (recursion/iteration),
2. whether subjects had to search over a list of names or over numbers, that is, whether they had to work with the book titles or the catalog numbers (list/number),
3. whether the answers were accumulated in a forward or reverse manner when doing the recursive call or iterative loop (forward/reverse),
4. whether success or failure instances were accumulated in the answer, that is, whether a book title did or did not belong to a certain category (success/failure), and
5. whether the answer was returned as soon as the accumulation test was negative or whether the function worked its way through the whole database by skipping negative instances (eject/skip).

The last four task dimensions were selected because they interacted with the recursion/iteration dimension to produce variations of code with which students in programming courses experience difficulty. We hoped to get at the heart of these difficulties with a systematic study of the variations.

Figure 1 gives an example of a recursive function that works on a list, accumulates successful instances in a forward manner, and ejects on a positive nonaccumulation test. The example in Figure 2 is its iterative analog on numbers.

All functions were patterned similarly: The first line contained a function header, the second line a test for termination. Lines 3 and 4 carried out the tests and accumulated the appropriate answer according to the function specifica-

Figure 1. **A recursive function.**

The following is an illustration of a SIMPLE function called FIRST-SCIENCE which takes a list of book titles and returns an answer list that contains all science titles from the first of the list to the first nonscience title. The answer list contains the titles in the order they appear in the variable. For example, FIRSTSCIENCE [basic logic faust lisp dune] returns the answer [basic logic].

```
1. firstscience list IS
2. IF (list = [ ]) THEN [ ],
3. IF ((FIRST list) ISA? science) THEN
       ((FIRST list) PRE (firstscience (REST list))),
4. ELSE [ ].
```

Comments:

1. Header: function-name ("firstscience") and variable ("list").
2. Termination condition: If the list is empty, then return the empty list.
3. Conditional branch: If the first item of the list is a science title, then return the first item and put it in front of the result of the recursive call on the rest of the list.
4. Conditional branch: Else return the empty list.

Figure 2. **An iterative function.**

The following is an illustration of a SIMPLE function called FIRST-SCIENCE which takes a list of book catalog numbers and tests numbers less than and including the given number. The function returns an answer list that contains all science catalog numbers from the given number to the first nonscience number. The answer list contains the numbers in descending order. For example, FIRST SCIENCE 7 returns the answer [7 6], if number 5 is a nonscience title.

```
1. firstscience num IS
2. UNTIL (num = 0),
3. IF (num ISA? science) THEN
       (ANSWER BECOMES (num POST ANSWER)),
4. ELSE DONE,
5. REPEAT WITH (num BECOMES (SUB1 num)).
```

Comments:

1. Header: function-name ("firstsciene") and variable ("num").
2. Loop termination: Test if the number has reached 0.
3. Conditional branch: If the number is a science title, then include it at the end of the answer-variable.
4. Conditional branch: Else eject from the function.
5. Repeat statement: Repeat the loop with the number decremented by 1.

tion. For the iterative function, a fifth line contained the repeat statement. Of all the dimensions varied, the difference between recursion and iteration was the most profound on the level of the actual code. As can be seen by comparing Figures 1 and 2, it affected every line of code but the first.

Differences on the list/number and forward/reverse dimensions involved simple term replacement. When working with a list, the constructs FIRST

<list> and REST <list> were needed. For numbers, the analogical constructs were a number variable <num> and SUB1 <num>. The termination test was list = [] for the list and num = 0 for the numbers. Otherwise, the function remained unchanged when going from list to number.

The forward/reverse dimension only affected the answer accumulation. For recursion, the function word PRE signified forward accumulation, POST signified reverse accumulation. For iteration, POST signified forward accumulation, and PRE signified reverse accumulation. In both cases, the change from forward to reverse accumulation involved only changing one function word.

On the success/failure dimension, going from accumulation of success cases to accumulation of failure cases involved exchanging the then- and else-actions of Lines 3 and 4 of the code (see Figures 1 and 2) for each other. This was true in both iteration and recursion.

The eject/skip dimension required more subtle changes. In recursion, a function ejected the answer after the empty list, [], had been returned. A function skipped over cases when a recursive call was made without an accumulation action for the element the function currently was working on. For iteration, the function word DONE after a test made the program eject from a loop, while ANSWER BECOMES ANSWER was used for skipping the construct.

Instructions. The instructions about how to write the recursive and iterative functions emphasized three aspects. First, a theoretical explanation of the recursive or iterative construct was given. Then, a template for writing the function was presented. Finally, the subjects obtained an example of a recursive or iterative function.[1]

Design. There were two phases in the experiment, training and transfer. In each phase, some of the task dimensions were varied systematically. There were 16 conditions overall, obtained by having subjects transfer from programming any of number-iteration, number-recursion, list-iteration, or list-recursion in training to any of these in transfer. In each condition, there were two subjects. One subject did forward accumulation in training and reverse accumulation in transfer, while the other subject did the opposite.

Within each condition, subjects had to solve four tasks that were obtained by crossing the two task dimensions success/failure and eject/skip. The four tasks obtained in this fashion were given to the subjects in a fixed order. This order was kept constant over all subjects.

As an example, the subjects in the training condition recursion-list-forward obtained problems they had to solve recursively, by working on a list and by doing forward accumulation in the answer. Given this, the four problems they had to solve were ordered in the following way: First, subjects had to accumu-

[1] A transcript of the instructions can be obtained from the authors.

late successful instances while ejecting at the first positive nonaccumulation test (success-eject). The second task accumulated failure instances and had the same eject-condition (failure-eject). The third task also accumulated failure instances, but it worked through the whole list given, skipping the nonaccumulated elements (failure-skip). Finally, the fourth task accumulated successful instances and skipped nonaccumulated elements (success-skip). The same task sequence was employed in the transfer conditions.

Procedure. The participants in the experiment were Carnegie-Mellon University undergraduates who had little or no programming experience. The maximum programming experience allowed was one introductory programming course. The subjects were assigned randomly to conditions. Because it proved difficult to assess differences in programming proficiency prior to the experiment, we employed analyses of covariance (ANCOVAS) to control for differences in programming skills (see the Results section).

The whole experiment took place in two sessions over two consecutive days. In the first session, subjects were familiarized with the commands of the programming language and wrote four simple three-line function definitions which employed no recursive or looping structure. One of the tasks, for example, was to test if the first book title of a list belonged to a given category. This first session was identical for all subjects.

In the second session, the subjects obtained instructions according to the condition they were assigned. In the training phase, they learned either about recursion or iteration and then had to solve the four tasks described above. Each subject was given the first task he or she had to solve as an example. After completing the training phase, subjects went directly into the transfer phase. The procedure for the transfer phase was identical.

The experiment was self-paced. Subjects learned from an instruction booklet and did the tasks on a computer terminal. After typing in the code for a problem, the subjects could change, delete, or insert a line of code to correct errors they had noticed. After subjects signaled that they were done with a problem, their code was tested for correctness. If it was correct, the subjects got positive feedback; otherwise they got an error message and an example of a correct function which they could examine for as long as they wanted. Subjects did not necessarily have to write the same function as was presented to them, but most subjects chose to do so.

The learning criterion set for the subjects was to solve each of the four functions correctly. If they made a mistake, they had to rewrite the function. Subjects could solve the functions with the example present in their first pass. If they made any mistakes, the example function was taken away, and they had to solve the tasks on their own on their second pass. It took the subjects between 2 and 5 hr to finish the entire experiment. Nine subjects could not finish the experiment and gave up prematurely. The reasons for their failure ranged from

genuine difficulties in understanding the task to fatigue and insufficient time to finish the experiment. The data of two subjects were discarded because they showed unusually high differences between the training and the transfer phase (> 30 min). One of the subjects was faster in the training phase, the other in the transfer phase. Both of the subjects were in the recursion-iteration transfer condition. Additional subjects were run to replace these 11 subjects.

2.2. Results

The data collected included the time for each line of code written and a record of the subjects' terminal interactions, including all editing operations. This section will present results about overall response times and response times on individual tasks. Response time for each problem was defined as the time between the presentation of the problem and the typing of the final function terminator (a period), including all the editing operations a subject might have performed. The overall response times were the sums of all the individual problem response times.

An ANCOVA was performed on the overall response times with programming construct (recursion-recursion, recursion-iteration, iteration-recursion, iteration-iteration) and datatype (list-list, list-number, number-list, number-number) as between-subjects factors and phase (training, transfer) as within-subjects factor. The response times in the definition phase were used as the covariate because the phase captured subject attributes such as typing speed and basic understanding of programming and the SIMPLE language, but not skills related to recursion or iteration. The product-moment correlation between the times of the definition and training phases was .83, between the definition and transfer phase, .76. The ANCOVA yielded a main effect for phase, $F(1, 16) = 11.28, p < .01$, and a significant interaction between construct and phase, $F(3, 16) = 3.5, p < .05$. The main effect indicates that there was a general speed-up from the training to the transfer phase; the interaction indicates that the speed-up was not uniform over the four groups that worked with a different sequence of constructs. All other effects or interactions were not significant. Because the factor datatype was not significant, $F(3, 15) = 1.26, p = .33$, further results will be discussed with respect to the four groups obtained by crossing recursion/iteration with the training and transfer phases.

Figure 3 presents the total time to criterion and mean number of incorrect functions in the training and transfer phases for the four training-transfer conditions recursion-recursion, recursion-iteration, iteration-recursion, and iteration-iteration. The data are the actual time taken, not adjusted for the ANCOVA. It is obvious that the recursion-iteration condition behaved differently from the other three conditions. Post hoc comparisons revealed significant differences below the 5% level for tests between recursion-iteration and the three other conditions with respect to the total time taken in the transfer

Figure 3. **Experiment 1: Total time to criterion in seconds (means) and number of errors (in parentheses).**

Condition	Definition	Training	Transfer	Training/Transfer Difference
Recursion-recursion	1305	1799	1125	674
	(3.9)	(3.6)	(1.4)	(2.2)
Recursion-iteration	1702	2031	2244	−113
	(2.9)	(3.1)	(4.6)	(−1.5)
Iteration-recursion	1013	1852	1162	690
	(2.9)	(4.4)	(1.5)	(2.9)
Iteration-iteration	882	1270	908	362
	(2.1)	(1.1)	(1.6)	(−.5)
Note. $n = 8$ for each cell.				
Overall ($N = 32$)	1211	1738	1360	
	(2.9)	(3.0)	(2.3)	

phase. There were no significant differences between groups in the training phase.

The speed-up for the recursion-recursion and iteration-recursion conditions was significant, $t(7) = 3.23, p < .05$, and $t(7) = 3.09, p < .05$, respectively; the speed-up for the condition iteration-iteration fell just short of significance, $t(7) = 2.01, p = .085$. Because this condition was the fastest overall, the lack of significance for iteration-iteration might have been due to a ceiling effect. The recursion-iteration condition actually slowed down, although not significantly, $t(7) = 1.02, p = .34$. The difference in speed-up between the iteration-recursion and recursion-iteration conditions was significant, $t(14) = 2.91, p < .01$.

The results reported so far were obtained through the manipulation of between-subjects factors. We now want to turn to the results of our within-subjects manipulation. Recall that within each condition and phase, subjects had to solve four problems that were given to them in a fixed order. These problems were success-eject (Task 1), failure-eject (Task 2), failure-skip (Task 3), and success-skip (Task 4).

Figure 4 shows the mean time to criterion for these four problems, split according to construct group and phase. An analysis of variance (ANOVA) on the total time to criterion for each task, with construct as a group factor and task as a within-subjects factor, revealed a significant main effect for task in the training phase, $F(3, 84) = 7.50, p < .001$. The group factor and the interaction were not significant. Post hoc comparisons showed that the third task (failure-skip) took longer than the other three tasks (Task 1–Task 3: $t(31) = 3.15, p < .01$; Task 2–Task 3: $t(31) = 2.82, p < .01$; Task 4–Task 3: $t(31) =$

Figure 4. **Experiment 1: Mean time to criterion for single tasks (in seconds).**

TRAINING (n = 8 for each cell)				
Task	Success-Eject	Failure-Eject	Failure-Skip	Success-Skip
Recursion-recursion	429	340	705	324
Recursion-iteration	441	468	650	459
Iteration-recursion	335	510	641	379
Iteration-iteration	361	346	369	194
TRANSFER (n = 8 for each cell)				
Task	Success-Eject	Failure-Eject	Failure-Skip	Success-Skip
Recursion-recursion	240	401	277	196
Recursion-iteration	452	620	684	489
Iteration-recursion	175	259	509	218
Iteration-iteration	293	245	192	178

5.65, $p < .001$). There were no significant differences among the other three tasks.

For the transfer phase, the analogous ANOVA revealed a significant main effect both for group and task, $F(3, 28) = 6.40, p < .01$, and $F(3, 84) = 3.02, p < .05$, respectively. The interaction was not significant. The group main effect was due to recursion-iteration, which took significantly longer to respond than the other groups (see Figure 3).

The task main effect was caused by Task 3 taking significantly longer than Task 1 and Task 4 (Task 1–Task 3: $t(31) = 2.16, p < .05$; Task 4–Task 3: $t(31) = 2.72, p < .05$), and Task 2 taking significantly longer than Task 4 ($t(31) = 2.74, p < .01$). Although the other t tests for differences between tasks were not significant, the results give the following rank ordering of time to criterion, longest to shortest response time: Task 3 (failure-skip), Task 2 (success-skip), Task 1 (success-eject), and Task 4 (failure-eject). This ordering, however, might be slightly misleading, for if the groups are collapsed according to the criterion "same or different construct from training to transfer," the interaction of group and task becomes significant, $F(3, 90) = 2.88, p < .05$. Figure 4 shows that only the groups that change constructs exhibited the longer response times with Task 3. Thus changing constructs seemed to aggravate the difficulty subjects have with Task 3.

We also analyzed the times to write each line of code, but these analyses did not provide any significant results. The data seemed to reflect some individual differences in solution style, but there did not seem to be any systematic relation to our task manipulations. For example, some subjects took very long to write the first line of code, whereas other subjects spent a large amount of time editing. We will try to point out some systematic relations between tasks and solution styles in Section 4.

The error data was analyzed in the same way as the response time data but did not yield any significant results. However, an analysis of the distribution of errors within the first four trials of each phase (i.e., the subjects' first tries at each of the problems) showed some interesting patterns. In the training phase, the subjects working with the recursive construct made significantly more errors than the subjects working with the iterative construct (32 vs. 18 errors; $\chi^2(1) = 6.43, p < .05$). Also, subjects made significantly more errors on Task 3 than on the other tasks (20 vs. an average of 10 errors; $\chi^2(1) = 11.68, p < .01$). For the transfer phase, the four groups distinguished themselves significantly only on Task 3. Here, the groups that did not change constructs made significantly less errors on their first try than the groups that did change constructs (4 vs. 13 errors, $\chi^2(1) = 10.42, p < .05$).

We classified errors into semantic versus other, where semantic refers to programs that were syntactically correct and free of typing errors but for some reason did not do the right operation. Overall, 68% of the errors were semantic, with 52% involving problems getting programs to skip or eject.

2.3. Discussion

There seem to be two major effects responsible for the results obtained in this experiment. One is the difficulty the recursion-iteration group had in the transfer phase. This group showed longer response times and higher error rates in the transfer phase than any other group. The second effect is the difficulty subjects had with the skip/eject dimension in the training phase which re-emerges if they had to change constructs in the transfer phase.

There are two obvious features we can try to use to explain the strange behavior in the recursion-iteration condition. One is the greater conceptual difficulty of recursion. Perhaps dealing with recursion in some way hurt subjects when they came to iteration. The second is the greater syntactic complexity of iteration. Perhaps, subjects were not prepared to deal with the syntactic complexity after recursion. Before elaborating on either of these explanations, we will consider the result of Experiment 2, which tried to equate the syntactic complexity of recursion and iteration.

Because doing the first skip-problem was so difficult, it is interesting to examine what exactly the subjects had to do to solve the problem of skipping an element. In the recursive condition, skipping involved introducing a second recursive call without an answer accumulation after the test for negative instances. Subjects had never seen a function with two recursive calls in this experiment, nor had they been told explicitly that this was possible at all. Given the success-eject example, it required some creative effort and a good understanding of recursion to come up with the correct solution without being given the example. Only 4 out of 16 subjects succeeded in doing so in the training phase, and 1 out of 8 in the transfer condition iteration-recursion.

For the iterative skipping construct, there were two immediate solutions for

going from the example given (success-eject) to the skip case. One solution involved omitting the fourth line of code (ELSE DONE, see Figure 2). The loop would still work and would accumulate only successful cases. To get it to accumulate failure cases involved a second, rather idiosyncratic change. Because no negative test was available in SIMPLE and because there were only three categories, one could write two tests to accumulate the two categories other than the negated category. Only two subjects in the iterative conditions came up with this solution. Six more subjects actually generated the skipping construct that was used in the feedback given to the subjects, ANSWER BECOMES ANSWER, which left the answer variable unchanged if the test failed on a given iteration.

Given this interaction of the skip/eject and the recursion/iteration dimensions, it is a reasonable assumption that the subjects who changed constructs were exposed to most of the difficulties involved in the skip/eject dimension twice. There obviously is not much opportunity for transfer because changing constructs also altered the way the skip/eject dimension had to be written. Thus, exactly these subjects had difficulty with the failure-skip problem in transfer.

It should be emphasized that although subjects had to create something new to deal with the skip case, they were given sufficient information from which, in principle, they could create the program. They equally had to create something new to deal with the failure dimension and had much less difficulty. The basic point is that this dimension, which is concerned with flow of control, is harder to master.

It is also worth noting two null effects we obtained. First, there was no difference in the solution times between the iterative and recursive group in the training phase. In our view, the equal solution times seem to result from a trade-off between syntactic complexity in iteration and semantic difficulty in recursion. As noted before, subjects made more errors in their first pass at recursive functions compared to iterative functions. If we accept this as an indication that subjects had a harder time understanding recursion, the question becomes one of accounting for the comparatively long time it took for subjects to do the iterative problems. In order to give this account, it is necessary to look at the iterative construct at the level of code the subjects had to write. The iterative construct in SIMPLE is rather awkward and looks very complicated compared to a recursive SIMPLE function. This is reflected in the high number of edits with iteration (110 vs. 62 for recursion). If the number of edits is an indication of subjects' syntax problems, as opposed to comprehension problems, the trade-off hypothesis is supported: In recursion, subjects spent their time trying to understand what they are asked to do on the algorithmic level, while in iteration, subjects knew the algorithm but had a hard time getting the code right.

The second null effect deals with one of the other task dimensions tested in the experiment, list/number. This dimension was systematically varied because we expected it to be relevant to the subjects' representation of the pro-

gramming constructs. However, it seems that the kind of problems we used were sufficiently easy to understand with both lists and numbers. There was no operation required that did any mathematical operation other than subtracting, and the lists had all very simple one-level structures that did not require any complex list structure manipulations.

3. EXPERIMENT 2: A CLOSER LOOK AT TRANSFER ISSUES

This experiment is essentially a replication of the previous one. Although the basic design remained the same as in the first experiment, a few changes were made to improve its shortcomings. This included changes to SIMPLE, such as simplifying the iterative construct so that the syntactic complications present in the first experiment were reduced. Additionally, the possibility of a negative test was introduced, which further simplified the task of writing the functions. To get better data on the subjects' learning, we made the task harder by changing the example problem given to the subjects. In the first experiment, the first task subjects had to solve was given to them as an example. Now, the example was a function that demonstrated the recursive or iterative construct but was not one of the experimental problems.

Two further changes were made to the design of the experiment. Because the first experiment showed that list/number did not influence the results and we did not know the impact of the forward/reverse factor, we exchanged these two dimensions in the design. In addition, the task sequence within each of the two phases, which had been kept constant in the first experiment, now was varied to test for differences between the individual tasks.

With respect to the forward/reverse dimension, we expected subjects to slow down if they changed constructs, because changing from recursion to iteration or vice versa elicits a counterintuitive change in the forward/reverse dimension on the code level. If the construct value changes, subjects have to use the same function word (pre or post) to change the order of accumulation. On the other hand, if the construct value remains the same, subjects have to change the function word. This is not an artifact of the SIMPLE language, but involves the difference between recursion and iteration. Thus, varying this dimension provides an additional test for subjects' understanding of the programming constructs.

3.1. Method

Programming Problems. The tasks used were the same as in Experiment 1. Some features of SIMPLE were changed in order to simplify the language. The first two features dealt with the looping construct in iteration. The function word LOOP was used to indicate the beginning of an iterative loop, and the termination test for the loop was approximated as much as possible to the termina-

tion test for recursion. Manipulation of the eject/skip dimension was made substantially easier with the iterative construct. Skipping could now be achieved by simply doing nothing—the ANSWER BECOMES ANSWER construction of Experiment 1 became obsolete. The success/failure dimension could now be manipulated by exchanging only the function words ISA? and ISNOTA?. It was no longer necessary to exchange whole lines as in Experiment 1. Figure 5 shows an iterative function written in the new version of SIMPLE.

Instructions. The instructions were nearly identical to those in Experiment 1, with some changes made to accommodate the new features mentioned above. In addition, subjects were no longer shown an example of a task they actually had to solve. Rather, this time the example was a sorting function that exhibited all the features of the problems in the experiment except for the negative test (see Figure 6).

Design. The design was basically the same as the design of Experiment 1. There was a training and a transfer phase, and 16 between-subjects conditions. The only difference was the exchange of the factors list/number and forward/reverse. The factor forward/reverse now was crossed with recursion/iteration to obtain the four transfer conditions for each of the four training conditions. Thus, subjects transferred from programming any of forward-iteration, forward-recursion, reverse-iteration, or reverse-recursion in training to any of these in transfer. The factor list/number was always changed to the opposite of the training condition for each subject.

Within each of these conditions, subjects again were given the four problems obtained by crossing the factors success/failure and eject/skip. In this experiment, the sequence of the four tasks was varied according to the following criterion: In going from one task to the next, one and only one value on one of the two dimensions was changed. The eight sequences fitting this criterion were counterbalanced with the four between-subjects training conditions. The eight subjects in each of the four training conditions obtained a different task sequence. In the transfer phase, the four tasks for each subject were presented in the same sequence as in the training phase.

Procedure. The procedure was identical to that of the first experiment except that the two sessions of the experiment were held either one or two days apart, rather than on consecutive days. In addition, subjects were now allowed to keep the example problem and were free to work with it until the end of the experiment.

Subjects were Carnegie-Mellon University undergraduates with little or no programming experience. It took the subjects between 3 and 5.5 hr to go through the whole experiment. Seven subjects could not finish the experiment or gave up prematurely. Additional subjects were run to replace them.

Figure 5. **A new iterative function.**

The following is an illustration of a SIMPLE function called NOSCIENCE, which takes a list of book catalog numbers and tests numbers less than and including the given number. The function returns an answer list that contains all nonscience catalog numbers less than and including the given number. The answer list contains the numbers in descending order. For example, NOSCIENCE 7 returns the answer [7 6 2 1], if numbers 3 to 5 are science titles.

```
1. noscience num IS
2. LOOP,
3. IF (num = 0) THEN DONE,
4. IF (num ISNOTA? science) THEN
       (ANSWER BECOMES (num POST ANSWER),
5. REPEAT WITH (num BECOMES (SUB1 num)).
```

Comments:

1. Header: function-name ("nosciene") and variable ("num").
2. Start of loop.
3. Loop termination: If the number has reached 0, then eject.
4. Conditional: If the number is not a science title, then include it at the end of the answer-variable.
5. Repeat statement: Repeat the loop with the number decremented by 1.

Figure 6. **The new example function.**

The following is an illustration of a function called SORT, which sorts a list of book titles so that all science titles are at the beginning of the list. For example, Sort [dune faust zorba lisp] returns the answer [lisp zorba faust dune].

Recursive example:

```
1. sort list IS
2. IF (list = [ ]) THEN [ ],
3. IF ((FIRST list) ISA? science)
       THEN ((FIRST list) PRE (SORT (REST list))),
4. ELSE ((FIRST list) POST (SORT (REST list))).
```

Iterative example:

```
1. sort list IS
2. LOOP,
3. IF (list = [ ]) THEN DONE,
4. IF ((FIRST list) ISA? science)
       THEN (ANSWER BECOMES ((FIRST list) PRE ANSWER)),
5. ELSE (ANSWER BECOMES ((FIRST list) POST ANSWER)),
6. REPEAT WITH (list BECOMES (REST list)).
```

3.2. Results

The ANCOVA on the overall response times was done with programming construct and accumulation type (forward-forward, forward-reverse, reverse-forward, reverse-reverse) as between-subject factors and phase as within-subjects factor. The response times of the definition phase were taken as the covariate. The product-moment correlation between definition and training phase was .19, between definition and transfer phase, .26. These correlations are substantially lower than the ones obtained in Experiment 1. Two explanations for this can be offered. On the one hand, the problems in the training and transfer phases were more difficult to solve in Experiment 2 than in Experiment 1, while the definition phase remained the same in both experiments. On the other hand, the variance of the subject sample in Experiment 2 was lower than in Experiment 1. As in Experiment 1, analyses on error data did not produce any significant results.

The analysis of covariance on response times yielded a main effect for phase, $F(1, 16) = 27.48, p < .001$, and a significant interaction between phase and construct, $F(3, 16) = 4.14, p < .05$, signifying (as in Experiment 1), a general speed-up between training and transfer phase, except for the recursion-iteration condition. All other effects or interactions were not significant. Because the factor accumulation type was not significant, $F(3, 15) = 2.08, p = .15$, further results will be discussed in the same way as in Experiment 1, by refering to the four groups: recursion-recursion, recursion-iteration, iteration-recursion, and iteration-iteration.

Figure 7 presents the total time to criterion and mean number of errors for these four conditions. As in the first experiment, there were no significant differences between the conditions in the training phase. In the transfer phase, the following comparisons proved significant: recursion-recursion versus recursion-iteration, $t(14) = 5.67, p < .001$; recursion-recursion versus iteration-recursion, $t(14) = 2.40, p < .05$; and recursion-iteration versus iteration-iteration, $t(14) = 4.05, p < .001$.

It seems that changing constructs in the transfer phase had a more marked influence in Experiment 2. If analyzed in terms of whether groups changed constructs or not, that is, by collapsing the recursion-recursion and the iteration-iteration groups into a same-construct group, and the recursion-iteration and the iteration-recursion groups into a different-construct group, a t test of the transfer phase times was highly significant, $t(30) = 4.22, p < .001$. The group that changed constructs took longer overall to go through the transfer phase.

With respect to speed-up from the training to the transfer phase, the first experiment was nearly replicated: The conditions recursion-recursion, $t(7) = 8.55, p < .001$, iteration-recursion, $t(7) = 3.75, p < .01$, and iteration-iteration, $t(7) = 3.17, p < .05$, exhibited speed-up, while the recursion-iteration condition did not, $t(7) = -.27, p = .80$. Specifically, the difference

Figure 7. **Experiment 2: Total time to criterion in seconds (means) and number of errors (in parentheses).**

Condition	Definition	Training	Transfer	Training/Transfer Differences
Recursion-recursion	1193	2678	1225	1453
	(1.5)	(3.1)	(1.1)	(2.0)
Recursion-iteration	1512	2219	2329	−110
	(4.1)	(5.1)	(4.9)	(.2)
Iteration-recursion	1521	2893	1784	1109
	(4.2)	(4.1)	(4.2)	(−.1)
Iteration-iteration	1208	2385	1304	1081
	(2.9)	(3.6)	(2.9)	(.7)
Note. n = 8 for each cell.				
Overall (N = 32)	1358	2530	1608	
	(3.2)	(4.0)	(3.3)	

in speed-up between recursion-iteration and iteration-recursion was significant, $t(14) = 2.84, p < .05$.

Thus we have replicated the asymmetry in transfer despite efforts to equate syntactic complexity. It does appear that our changes to the iterative construct to make it of comparable syntactic complexity were successful. Subjects made almost the same number of edits with the iterative programs (158) as with the recursive programs (148).

We performed two ANOVAs for the within-subjects manipulation. One used task sequence as a factor, the other used task type. The first of these analyses, with task sequence as within-subjects factor and construct as between-subjects factor, showed a main effect of speed-up from the first task to the last task in both phases regardless of the construct subjects worked with (training: $F(3, 90) = 31.81, p < .001$; transfer: $F(3, 90) = 18.79, p < .001$). Although the group factor was not significant in the training phase analysis, it was in the transfer phase analysis, $F(1, 30) = 13.51, p < .001$, reflecting the fact that groups that changed constructs took longer on each of the four tasks than groups that did not change. The interaction was not significant in either phase.

In the first experiment, because task type and task sequence were confounded, it was unclear whether the two task dimensions manipulated within the sequence of tasks were contributing to the variance of the response times. In the current experiment, we separated these two factors. An ANOVA for each phase with group as between-subjects factor and the four task types as within-subjects factors revealed no significant effects. Thus it can be assumed that none of the four tasks was more or less difficult than any other. Furthermore, the task dimensions success/failure and skip/eject seemed to be equally difficult in recursion and iteration.

Although the four tasks were of equal difficulty, it is still possible that a change in the value of the dimensions skip/eject or success/failure could have caused difficulties. Recall that, as subjects progressed from one task to the next, only one of the binary values on the two dimensions skip/eject and success/failure was changed. Because there was no overall difference between the tasks, tests for sequence effects depending on changes on one of the two task dimensions could be performed. Half of the subjects changed values on the skip/eject dimensions from Task 1 to Task 2, while the other half changed values from Task 2 to Task 3. The same is true for the success/failure dimension. Task 1 and Task 4 had all subjects in the same condition with respect to experienced value change. Thus, the crucial differences should be found with response times to Tasks 2 and 3. Figure 8 presents the times to criterion according to this analysis.

The interaction between point of change for the skip/eject dimension (whether subjects changed on Task 2 or 3) and task sequence was significant (training: $F(1, 30) = 6.06, p < .05$; transfer: $F(1, 30) = 16.19, p < .001$). The main effect of point of change was not significant. The interaction is due to a significant difference in solution time on Task 2. The group that changed the value on the dimension skip/eject on Task 2 took longer on this task than the other group (training: $t(30) = 2.34, p < .05$; transfer: $t(30) = 2.39, p < .05$). In addition, only the change-on-2 group exhibited speed-up from Task 2 to Task 3 (training: $t(15) = 3.08, p < .01$; transfer: $t(15) = 2.79, p < .05$). The change-on-3 group actually slowed down, although significantly only in the transfer phase, $t(15) = 3.06, p < .01$.

In doing this analysis for the skip/eject dimension, we implicitly have done it for the success/failure dimension, too. Remember that changes on the success/failure dimension were complementary to changes on the skip/eject dimension: If one dimension changed its value from one task to the next, the other did not. Thus our results show that subjects were faster overall if the dimension success/failure changed from one task to the next, while they slowed down if this dimension did not change. Taken together, this implies that the skip/eject change was the more difficult factor.

As in the previous experiment, most of the errors could be classified as semantic (68%), and a large fraction of these (36%) involved errors with skip/eject. The next most common semantic error concerned confusions of success and failure (22%).

3.3. Discussion

The pattern of results obtained in Experiment 2 replicates the major outcomes of Experiment 1. Subjects had problems transferring from recursion to iteration and difficulty with the skip/eject dimension. The complexity of the iterative syntax is not the reason for the peculiar behavior of the recursion-

Figure 8. **Experiment 2: Mean time to criterion for trial sequence (in seconds), with groups divided according to location of first change on the skip/eject dimension.**

Trial	1	2	3	4
TRAINING				
Change2 (n = 16)	1159	657	413	359
Change3 (n = 16)	1116	465	534	384
TRANSFER				
Change2 (n = 16)	695	495	277	278
Change3 (n = 16)	637	278	440	214

Note. Change2 = Group changes value on the skip/eject dimension on Trial 2. Change3 = Group changes value on the skip/eject dimension on Trial 3.

iteration group, as this syntax was simplified in Experiment 2. As a matter of fact, there is nothing in the data of the second experiment that hints at an explanation for the difficulty experienced by this group.

To return to our original questions, this experiment showed that subjects actually can acquire the skill of writing recursive functions within a few hours of instruction, thus confirming the result of Anzai and Uesato (1982). In addition, it was shown that subjects could acquire the recursive construct as easily as they acquired the supposedly more natural iterative construct. When subjects did learn both constructs, the sequence of acquisition was crucial. When the subjects learned iteration first, and transferred to recursion, their speed-up was significant. When they started with recursion, there was no speed-up after they were transferred to iteration.

The claim of Anzai and Uesato that knowing iteration facilitates learning recursion is not really supported by these results. Our control group, which practiced recursion only, arrived at the same performance level in the second phase as did the iteration/recursion group. Therefore, all that can be said about the effect of learning iteration before learning recursion is that it seems to be as effective as increased practice on recursion itself. This, of course, does not change the pedagogical implication that it is beneficial to learn iteration before learning recursion.

In order to understand the peculiar behavior of the recursion-iteration group, it is important to look at the kind of problem solving the subjects actually were doing in this experiment. As Pirolli and Anderson (1985) showed, if the instructional environment permits, subjects generally solve their first recursive problems by analogy to examples. The instructions given in this experiment did provide an example and so the analogy process was available. Subjects' informal comments suggested they did make a good deal of use of analogy to the example. It is possible that they solved the recursive problems by analogizing

from the surface structure of the example without really understanding how it worked.

If this were true, we could hypothesize that the recursion-iteration subjects did not really learn anything about the control construct in the training phase. Rather, they learned a set of surface templates by analogy. If subjects did not understand flow of control in recursion, they would have no basis for any transfer to a new construct. Therefore, the null transfer from recursion to iteration may be because subjects actually had to learn the iterative construct in the transfer phase as if they had had no previous experience with the task. This explanation could account for the lack of speed-up in our recursion-iteration subjects. In addition, it fits in with the fact that their error data in the transfer phase is not significantly different from the error data of subjects who learn iteration in the training phase.

Our third initial question was how other factors influence the acquisition of control constructs. Of the task dimensions we varied, only the skip/eject dimension proved to have any influence. Changing on this dimension was difficult and interacted with the recursion/iteration dimension. Although we cannot be sure if this is an artifact of our particular problems and/or programming language, it is interesting to note that the skip/eject dimension is the one dimension besides recursion/iteration that most influences flow of control.

4. PROTOCOL ANALYSES

In order to test our speculations about subjects' representations and strategies, we collected protocols of four subjects in the recursion-iteration and iteration-recursion conditions. In taking the protocols, the procedures of Experiment 2 were followed. The only difference was that the experimenter stayed in the room while the subjects solved the problems. His task was to prompt the subjects if they kept quiet for a long period of time. Otherwise, he did not interfere with the subjects' behavior. The experimenter's presence had one important consequence, however. Although subjects in Experiment 2 were told they could ask the experimenter for help if they did not understand the instructions or the feedback given to them, they hardly ever did so. The protocol subjects, on the other hand, were much more likely to ask questions. In these cases, the experimenter reiterated the information given in the instruction until the subjects decided that they had received a satisfactory answer. As a result, the protocol subjects showed less random behavior than some subjects in Experiment 2 (i.e., they never copied a previous function to solve the next problem, and they never went right to the feedback without trying to solve the problem).

There were eight protocols overall, four from subjects in the recursion-iteration condition (Subjects 1 to 4) and four from subjects in the iteration-recursion condition (Subjects 5 to 8). All subjects worked with lists in the train-

ing phase and with numbers in the transfer phase. A comparison with the data of Experiment 2 shows that all iteration-recursion subjects approximated the behavior of the subjects in the same condition of Experiment 2. Mixed results were obtained with the recursion-iteration subjects. One subject (S4) slowed down, while another one (S1) exhibited a slight speed-up. The other two subjects (S2 and S3) did not finish the transfer part of the experiment, although they finished their third and fourth trials, respectively, in the transfer phase. In both cases, subjects were unwilling to cooperate further due to fatigue. All subjects in the recursion-iteration condition took longer overall than would have been expected from Experiment 2.

There are two reasons why we did not exclude Subjects 2 and 3 from the analysis. First, except for taking longer, they did not exhibit any obvious behavior differing from that of Subjects 1 and 4. Second, we concentrated our analysis of the protocols on the first four problems in each phase. It became clear from the protocols that subjects had established their problem-solving strategy at about the third or fourth trial, as discussed in the following section.

4.1. General Characteristics of the Protocols

The protocols of both groups show some common characteristics supporting the findings of Anderson, Farrell, and Sauers (1984) in their analyses of protocols from subjects learning to program in LISP. In general, the knowledge subjects had acquired prior to the training phase of the experiment was not sufficient to allow them to solve the programming problems directly. The knowledge consisted of three parts: practice in the basic commands of the SIMPLE language; instruction about the construct, including a template showing the structure of the resulting function; and an example. When starting out, the main information that was used was the example that was given with the instructions. Subjects never made any reference to the template and frequently did not remember basic SIMPLE commands, the knowledge of which they had demonstrated earlier.

As subjects went through the training phase, they usually became more accurate and faster in solving the problems. After two or three exposures to the feedback consisting of correct functions, subjects were able to keep these correct functions in working memory and to adjust them to the given problem. References to the example were made only for syntactical questions such as parenthesization and argument ordering.

For the most part, errors committed after Problem 3 were mere lapses, (e.g., forgetting a comma or mistyping a word) or working memory failures (e.g., misremembering the previous function and falsely copying it). Subjects seemed to have developed a successful procedure for doing the problems at that point.

This general characterization of the training phase also holds for the transfer phase. That is, subjects showed reliance on the example in Trial 1, developed a

solution procedure over Trials 2 and 3, and finally made a smooth application of this procedure.

Differences in the protocols obtained from the recursion-iteration and iteration-recursion groups show that subjects differ in their use of the example, and, more generally, in the way they approach the problem-solving process. These differences are examined in detail in the following section. Although such protocol evidence is necessarily opportunistic, we would like to argue that these dissimilarities account for the observed performance differences between the groups in the transfer phase. This account will schematize the protocols, and thus make them look more similar than they really are, but we feel that it captures the major differences that can be detected between the two groups. In order to support the hypothesis about different solution strategies given in the next section, we later present some quantitative analyses about utterances in the protocols which support our account.

4.2. Transferring From Recursion to Iteration

A typical solution process in the training phase was given by Subject S4. When solving his first problem, the subject was able to map the first three lines of code from the example to his solution. Although he was not very verbal about the mapping process, he took considerably more time to write the first three lines of code than would be necessary for straightforward copying. Rather, he tried to check each line of the example with regard to its relevance to the problem specification. He was able to convince himself that he could take over the first three lines of code without modification. This solution process broke down for the fourth line of code. There, the subject clearly stated what he had to do—skip the nonscience elements—but he also remarked that he did not know how to do it. Because the example did not provide a clue here, the subject finally tried a construction that he did not expect to work.

The reliance on the example varied markedly between subjects, as shown by other protocols from subjects in this group. One subject (S1) pursued a stubborn word-by-word mapping from the example to the problem. She went through several iterations of trying to understand the example and mapping it to the problem. Although she pursued a variety of dead ends, she finally succeeded in getting the function right by finding a mapping for each piece of code in the example. Another subject (S3) virtually ignored the example because she was convinced that she did not understand it. She tried to solve the problem by applying the knowledge she extracted out of the instructions. Although she did not succeed in solving the problem in this way, she maintained this strategy throughout the training phase. Her improvement was due to repeated exposures to the correct functions which she could approximate better after each trial.

What unites these different ways of using example and feedback is the fact

that none of the subjects in this group demonstrated any sign of abstracting information about the flow of control from the example and/or the problem solutions. Their reasoning remained at the surface level of actual code; their strategy could be characterized as a form of means-ends analysis. Subjects tried to detect the difference between the problem and the example or the previous solution shown to them and then tried to reduce this difference doing a local repair consisting of exchanging a small part of the code.

What the subjects seemed to take away from the training phase was a strategy that proved reasonably effective, although it neither required nor created an understanding of the construct. Most subjects initially used the same approach in the transfer phase. Three of the four subjects tried to solve the first iteration problem by mapping from the example. Because the example was a function working on lists and the subjects now had to work with numbers, none of the lines, except for the loop statement, could be copied directly. Although all subjects failed to write this first problem correctly, they seemed to grasp the iterative control structure much better than the recursive structure once they received feedback.

In order to provide a demonstration of how subjects neglected consideration of an algorithm and stuck to the code when doing the iterative problems, Figure 9 gives an excerpt from the protocol of S1 coding the second problem in the transfer phase. It illustrates how the subject used the example and her memory of the previous problem to do a line-by-line generation of the code, without being able to devise a working algorithm before she started coding. Although the subject demonstrated an intuitive understanding of the control structure, she still relied on the step-by-step approach carried over from the training phase. Each line of code was treated as a separate entity, without consideration of the relationship between the code and the control structure. The claim is made here that this coding strategy is typical for the subjects in this group. After presenting an analysis for the iteration-recursion group, more quantitative evidence will be presented to support this claim.

4.3. Transferring From Iteration to Recursion

Subjects starting out with iteration also made heavy use of the example in the first problem. With increasing experience, the subjects became more independent of the example and used it mostly for reference to syntax problems. A good example is Subject S6 doing the second problem in the training phase. She started out immediately writing the first two lines, then went through the loop mentally before she wrote the next line. She made the decision which test to use and wondered how to accumulate and where to put the repeat statement. Only then did she start writing out code. All this was done without reference to the example, which led her to write a false repeat statement, the only mistake in her code.

Figure 9. **Protocol excerpt: Subject 1, transfer phase, Trial 2.**

Problem Statement:
Write a function that takes a science ID as a variable. The function should return an answer list that contains all science IDs less than and including the variable. Answer list IDs should be in descending order.

1. "The first thing that we're doing here is function, and then the number . . ."
   Writes: func num is
   loop,
2. "and then we have our all purpose if . . ."
   Writes: if (num = 0) then done,
3. "Alright, if number is a science—if the number, yes, is a science then the answer becomes number, and while I'm in descending order, that would be post answer—yes I think."
4. "If number, now number is a function, right. So, number—I'm not sure about the syntax of this. Does it have to have . . . take an argument? I don't know. Am I supposed to have a paren there? I don't think so."
5. "Then—if the number is a science then answer becomes—that would be the number—. . . so it's number which is the function, and then pre answer."
6. "So is it pre or post? Pre."
7. "Alright, I'm rereading the instructions. . . . Answer becomes the number displayed in front of the answer. I don't really understand all that, but I'm going to continue."
   Writes: if (num isa? science) then (answer becomes (num (pre answer))),
8. "Now I have to do the sub1. No, no I need an else. If the number is not a science, that's my else."
9. "Now at the last time at this point I just said done."
10. (Rereading the instructions) "I wonder if this means it contains only science IDs. Alright, so I can't say else done, I have to say take out the first number, which would be first num, that would be answer becomes . . ."
11. "Alright, the answer becomes. . . . Now this one, the example gives everything back. I don't want it to give everything back. I want it to take that out."
12. "Now I'm not sure if this is gonna do that. If the number is a science then—answer becomes and is displayed—that's very circular. Else, if it is not a science, you extract it."
    Writes: else (answer becomes (first num))
13. "Now I am at the repeat. At this point, I've forgotten how to write this, but I want to tell the number, and I'm hoping this is right. Here I use sub1. So I am telling it to subtract number. The parens are wrong. I didn't put sub1 in its parens."
14. "Now . . .," (rereading the instructions) "I think I'm supposed to have—I wonder if I'm supposed to have pre answer in there. It doesn't say that in the example."
15. "I don't know how these things are arranged. We are assuming that these things in the catalog are arranged in descending order. If they are, this will skip over anything that is bigger. I expect it would go look for the next sub1. Well, I'm not going to worry about that."
17. "Repeat—now I have copied that pretty exactly, but I don't have any idea if this is going to work. I think this is just another approximation, so I'm gonna tell it to stop."
    Writes: repeat with (num becomes (sub1 num))

The other subjects in this group showed the same kind of planning behavior, with decreasing reliance on the example. Some of the subjects actually committed syntactic mistakes because they did not check the example for the right application of such statements as REPEAT WITH . . . or ANSWER BECOMES . . . .

There is some evidence that subjects carried over their problem-solving strategy into the transfer phase. The subjects in this condition seemed to use planning instead of means-ends transformation of the example during the transfer phase. The following quotation from S5's protocol illustrates this strategy transfer:

> We don't need a loop command here, we need an "if." We're gonna analyze from *v* down, we're gonna check to see if *v* equals 0—yeah, you can check for *v* equals 0.
>
> Let's see, you get to something that is a science [reading the instructions]. This program is gonna end when you hit the first science ID. How do I end this program? It's gonna end when you reach either a science or 0.
>
> I could make a list from *v* to the first science. If it isn't a science then you can add it in. If it's not, you don't. I'll take a wild stab at "if v isnota?" But you need something if *v* is 0. What if *v* equals 1? No that makes sense, I think. If *v* equals 0, then nothing."

When making these comments, the subject had only written the header line of the function. He sketched the algorithm, and only afterwards went on coding the function. Although he did not write a correct function, his strategy carried over to subsequent problems.

As will be discussed in the next section, there is no evidence that the understanding of recursion was better for this group than for the other group. It seems that the groups distinguished themselves mainly in the strategy they were using.

4.4. Quantitative Analyses

In order to substantiate the claims made in the previous sections, we looked for more quantitative indicators in the protocols that would help support our hypotheses. There are two major claims:

1. The groups use different strategies in solving the programming problems.
2. The groups acquire essentially the same mental models of the constructs.

We characterized the different strategies as means-ends analysis involving a line-by-line mapping for the recursion-iteration group and planning for the

iteration-recursion group. Apart from the anecdotal evidence in the protocols, one important indicator of this strategy difference should be the number of references subjects made to the example and the feedback (the correct functions). We would expect the recursion-iteration group to make substantially more use of these knowledge sources, because subjects did not rely on the control structure of the problems. Their mapping needed the feedback or the example as a point of departure. On the other hand, the iteration-recursion group needed the example primarily to check for syntactic errors. However, they also could use the instructions for this purpose. Furthermore, syntactic checking was not done very frequently, probably because the subjects had enough practice with the expressions they had to use. For these reasons, the recursion-iteration group should show more references to the example than the iteration-recursion group. Figure 10 shows the number of references made by both groups in the training and transfer phase. Only the first four trials in each phase have been considered because the subjects had learned the important features of the problems. An ANOVA with group as between-factor and phase and trial sequence as within-factors revealed that the only significant effect was the effect of sequence, $F(3, 18) = 10.81$, $p < .001$. Not surprisingly, most references occurred in the first trial in both phases. Although the test is not powerful enough to detect a significant difference with only four subjects in each group, $F(1, 6) = 2.16, p < .2$, the recursion-iteration group seemed to make more references to the example regardless of the phase. This would support our original conjectures.

Finally, we can look at one more indicator in order to find out about the subjects' representation of the control structures. This is the number of references to the control structure (loop or recursive call). As Figure 11 shows, the subjects did not distinguish themselves in the number of references over the first four trials in each phase. Again, only the trial sequence effect was significant in the ANOVA, $F(3, 18) = 7.89, p < .01$. This time, however, no trend is visible in the data, except for the fact that there were more references to the recursive control structure than to the iterative structure in both groups, $F(1, 6) = 3.99, p < .1$.

In order to understand subjects' representation of the control structure, it is not enough to consider how often they mentioned something about the structure. It is equally important to find out what exactly subjects were saying about their notion of control. Figures 12 and 13 give some sample comments from subjects in the two groups. With the iterative construct (Figure 12), all of the subjects' utterances were fairly straightforward. Subjects usually mentioned that they needed a **REPEAT** statement, and they became more confident about the loop structure as they proceeded through the experiment. There was no case in which their utterances about iteration revealed a misconception about what was going on in the program.

With the recursive construct, the picture looks different. None of the subjects volunteered a comment that unequivocally showed an understanding of recur-

Figure 10. **Number of references to the example function.**

Condition	Recursion-Iteration	Iteration-Recursion
TRAINING (n = 4 for each cell)		
Trial 1	17	8
Trial 2	10	0
Trial 3	3	0
Trial 4	3	0
Total	33	8
TRANSFER (n = 4 for each cell)		
Trial 1	13	6
Trial 2	6	1
Trial 3	5	0
Trial 4	2	0
Total	26	7

Figure 11. **Number of references to the control construct.**

Condition	Recursion-Iteration	Iteration-Recursion
TRAINING (n = 4 for each cell)		
Trial 1	6	5
Trial 2	4	1
Trial 3	4	2
Trial 4	0	0
Total	14	8
TRANSFER (n = 4 for each cell)		
Trial 1	3	6
Trial 2	1	4
Trial 3	2	1
Trial 4	1	1
Total	7	12

Figure 12. **Sample utterances about control structures: Iteration.**

Subject 1, Trial 1, Transfer:
"I have just said 'rest list.' Now, if the first list is a science, then it's at the beginning. And here I told it to stop if the first is not a science. So, now I want it to iterate, repeat with—"
Subject 1, Trial 4, Transfer:
"And then we have our repeat . . . Alright, that tells it to continue . . ."
Subject 4, Trial 2, Transfer:
"If I do an 'if' and then an 'else' and then another 'then,' is it going to include all—oh wait, there's also a 'repeat with.' "
Subject 5, Trial 3, Transfer:
"I have to reassign the answer, and 'x becomes' will be in the repeat loop. I'm gonna do repeat to chop it off."

Figure 13. **Sample utterances about control structures: Recursion.**

Subject 1, Trial 1, Training:
"Now, the trouble with 'rest list'—wait a minute, 'rest list' says it also gives the rest of the list. So maybe I don't want that there. I have to have that there, otherwise it will stop, it won't be recursive. 'Pre first list'—I wonder if what I want is just 'pre func rest list.' I never did understand what that meant."
Subject 1, Trial 3, Training:
"I want it to list the isnota's. Take a look at the rest of the list, exercise function, and put this at the beginning, so . . ."
Writes: then ((first list) pre (func (rest list))),
"else—now, if the first thing is not, is a science, then—if the first item in the list is not a science, put it first; however, if it is a science, I don't want that 'post,' what I want is exercise your function on the rest of the list. . . . So, do I want 'func rest list?' If the first line is not a science, then take it out, look at the rest of the list, exercise your function, and put this at the beginning. If it is a science, then look at the rest of the list and exercise your function. You don't want 'post,' no, no, no, so I think it's just . . ."
Writes: else (func (rest list)).
Subject 2, Trial 1, Training:
(looking at the example function "sort")
"I thought sort was a function in the language. Is sort already in there?"
Subject 2, Trial 2, Training:
"I need to use the function that I declared in the problem somewhere."
Subject 4, Trial 1, Training:
"If it's a science, then I'm gonna do like the example, put it in front and then continue 'funcing' it with the rest of the list."
Subject 5, Trial 1, Transfer:
"Wait, this is supposed to be recursive, I should have 'function v' in there somewhere. . . . I need to recur in there somewhere, I'm not gonna use 'answer becomes.' "
Subject 6, Trial 1, Transfer:
"Okay, then I put x before 'function sub1 x'—but is that gonna put me back up where 'sort x' is?"
Subject 8, Trial 2, Transfer:
"If 'num isnota? science,' it would take 'pre func sub1 num,' so it would go one lower, and go through the function again, and put that not-a-science before. I think that's the basic idea, but I'm not sure if it's gonna work."

sion. As Figure 13 demonstrates, the majority of utterances about recursion showed only a superficial level of understanding or a complete misconception. All comments were made by the subjects in the course of their problem solving. The subjects usually did not reflect on their understanding of the construct and were not asked about it. One subject (S1), however, volunteered the following statement about her understanding of recursion after she had correctly solved all problems in the training phase:

```
Code: func list is
          ......,
          if (first list) isa? science then [],
          else ((first list) pre (func (rest list))).
```

> I never did understand recursion. . . . If it's not a science, take it out, look at the rest of the list, and then I would think it would see this word, func, and it would say, "hey, you never told me what func is—you mentioned it up here in Line 1, but you never have spelled anything out."

Taken together, the evidence shows that subjects do not develop a correct representation of recursion. However, over the course of solving the problems, they can develop a working model that incorporates the notion of repeating some action over and over. This, of course, is the misrepresentation of recursion as a loop structure that was observed by Kahney and Eisenstadt (1982) and Kurland and Pea (1983).

5. GENERAL DISCUSSION

In this study, we established that there is asymmetric transfer between the acquisition of recursive and iterative control structures. Although there is positive transfer from iteration to recursion, there is no transfer from recursion to iteration. The protocol studies pointed out that this asymmetry may be due to differences in the solution strategies between subjects in the iteration-recursion and recursion-iteration conditions. Our subjects had further difficulties with the skip/eject dimension which also is closely related to flow of control.

With the following account of the differences in the solution processes between the two groups, we try to provide an explanation for the observed differences in solution times. The iteration-recursion group was able to extract the idea of a general notion of control in the training phase. When subjects went into the transfer phase, they were able to accommodate the recursive function to the representation of flow of control they had developed. Their previous experience with the task dimensions made it easier to distinguish the parts of recursive code responsible for the different actions to be taken. In particular, they were able to recognize the recursive call as a type of repeat statement. This account by no means presupposes that the subjects actually understood recursion. However, looking at a function in terms of flow of control enabled subjects to plan a solution. This strategy proved more effective than the one the recursion-iteration subjects settled on.

The recursion-iteration group started out by solving the problems using means-ends analysis. Subjects took the example and tried to match it line-by-line to the problem specification. With the constant feedback given after incorrect solutions, this amounted in the end to a memory task. If the relevant pieces of code could be remembered correctly, these subjects were guaranteed a solution. Subjects never needed to look at the function as a whole; they could concentrate on the small differences between the skip/eject or success/failure dimensions. And even if they took the whole function into account, it was unlikely that they followed the flow of control through the recursive function. When en-

tering the transfer phase, these subjects continued to use this strategy. They were not able to plan their functions on the level of the algorithm, because they had not acquired any representation of flow of control. Rather, they relied on the means-ends analysis method for local fixes. Because the surface form of the iterative code was sufficiently different from the recursive code, there was not much opportunity for transfer. Therefore, subjects had to learn how to deal with the new construct both without benefiting from their previous problem solving and with an inappropriate learning strategy.

Clearly, subjects do have difficulty with concepts involving flow of control. In either iteration or recursion, in training or transfer, subjects had difficulty mastering the skip/eject dimension. These results reinforce the importance of having adequate mental models of programs and, in particular, flow of control for mastering programming. Having a model of iteration enabled subjects to see how to transfer their training knowledge to recursion. In contrast, subjects studying recursive programs without the aid of such models were overwhelmed by the surface differences between iteration and recursion.

In general, we speculate it is good to teach iterative programming before recursive programming for students who have had no prior programming experience with either construct. Students can develop mental models of iterative procedures relatively easily. These mental models can serve as the basis for understanding recursive procedures—although not in the simple way our subjects did. Rather, recursive procedures can be understood in terms of a procedure looping over a stack of function calls. This is the copy model advanced by Kahney and Eisenstadt (1982).

It is informative to compare the results of our experiments with other studies of how people learn to program (Anderson et al., 1984; Pirolli & Anderson, 1985). All accounts emphasize the importance of analogical reasoning as a basis for learning to program. We characterized this process as a complex mapping procedure which goes beyond mere copying of an example. Although it has not been the purpose of this article to discuss analogical reasoning as a basic mechanism of skill acquisition (see Carbonnell, 1983; Gentner, 1983; Gick & Holyoak, 1980, 1983; Rumelhart & Norman, 1981), it should be noted that our experiments demonstrated the benefits as well as the limitations of the use of analogies. The benefits were clear in cases in which the subjects had developed an adequate, if sometimes incorrect, mental model of the programming construct. Here, subjects could make use of the example to write syntactically correct code. In accordance with previous studies, these subjects also demonstrated great savings in solution time when going from the first problem to the second. In addition, these subjects showed less reliance on the example in subsequent problems. On the other hand, the subjects who did not develop an adequate mental model of the programming construct used a mapping strategy that depended heavily on the surface features of the example through both phases of the experiment. As we have shown, this second group of subjects showed

no benefit of training in the transfer condition, although the first group did. This extends the observations of Pirolli and Anderson (1985), who showed that successful learning of recursion depended on an adequate mental model. We now know that possessing an adequate mental model is also critical to the transfer of learning between recursion and iteration.

In conclusion, the reason for programming novices' difficulty in understanding flow of control lies in their inability to develop adequate mental models of the task. The importance of mental models for the understanding of human skill acquisition is currently a well-represented research topic in cognitive science (see Gentner & Stevens, 1983). The preceding account tried to illustrate some of the cognitive processes underlying the formation of mental models of flow of control in programming, especially in relationship to the control dimensions of our tasks, recursion and iteration. We could demonstrate that these task variables influenced the development of mental models of our subjects.

Acknowledgments. We thank Peter Pirolli for his help in designing the programming tasks and for making his instructions on recursion and iteration available to us.

Support. This research is supported by contract N00014-84-K-0064 from the Office of Naval Research.

REFERENCES

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8,* 87–129.

Anzai, Y., & Uesato, Y. (1982). Learning recursive procedures by middleschool children. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society,* 100–102.

Carbonnell, J. G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Carbonnell, & T. M. Mitchell (Eds.), *Machine learning—An artificial intelligence approach* (pp. 137–162). Palo Alto, CA: Tioga Publishing.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science, 7,* 155–170.

Gentner, D., & Stevens, A. L. (Eds.). (1983). *Mental models.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology, 12,* 306–355.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology, 15,* 1–38.

Kahney, H., & Eisenstadt, M. (1982). Programmers' mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society,* 143–145.

Kurland, D. M., & Pea, R. D. (1983). Children's mental models of recursive LOGO programs. *Proceedings of the Fifth Annual Conference of the Cognitive Science Society,* 1–5.

Pirolli, P., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39,* 240–272.

Rumelhart, D. E., & Norman, D. A. (1981). Analogical processes in learning. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 335–360). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Shrager, J., & Pirolli, P. L. (1983). *SIMPLE: A simple language for research in programmer psychology* [Computer program]. Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology.

HCI Editorial Record. First manuscript received July 23, 1985. Revision received March 17, 1986. Accepted by Bill Curtis. Final manuscript received July 28, 1986. — *Editor*
